

UNIVERSITY OF
ILLINOIS LIBRARY
AT URBANA-CHAMPAIGN



CENTRAL CIRCULATION AND BOOKSTACKS

The person borrowing this material is responsible for its renewal or return before the **Latest Date** stamped below. **You may be charged a minimum fee of \$75.00 for each non-returned or lost item.**

Theft, mutilation, or defacement of library materials can be causes for student disciplinary action. All materials owned by the University of Illinois Library are the property of the State of Illinois and are protected by Article 16B of Illinois Criminal Law and Procedure.

TO RENEW, CALL (217) 333-8400.

University of Illinois Library at Urbana-Champaign

DEC 01 1999

When renewing by phone, write new due date below previous due date.

L162



Digitized by the Internet Archive
in 2013

<http://archive.org/details/numericalsoftwar969gear>

5/0.84
26r
e. 969
op-2

Math.

8

UIUCDCS-R-79-969
UILU-ENG 79 1715
COO-2383-0059

NUMERICAL SOFTWARE: SCIENCE OR ALCHEMY?

by

C. W. Gear

June 1979

THE LIBRARY OF THE

JUN 29 1979

UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-79-969

NUMERICAL SOFTWARE: SCIENCE OR ALCHEMY?

by

C. W. Gear

June 1979

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

Supported in part by the U.S. Department of Energy, Grant US ENERGY/EY-76-S-02-2383.

1. INTRODUCTION

"Numerical Software" is a term that is used rather liberally today to describe a range of activities. In this talk I want to address the questions: "Is there anything being done under the heading Numerical Software that was not done in past years when we just called it programming?", "Are those things being done important?", and "Are they a science?" First, I will look at the nature of numerical software and then discuss what is particularly difficult about it. The third part of the talk briefly examines the science behind such software, and finally we will look at the the areas where that science does not help us.

Numerical software production is viewed by many people either as a routine programming task or as a by-product of that dull subject, numerical analysis, which itself falls somewhere between mathematics and computer science, too applied for the one and too irrelevant to the other. However, I want to show that there is a significant difference between the concern and approach of either a numerical analyst or of a programmer on the one hand, and a person who writes numerical software on the other. I will call the latter person a "numerician" for want of a better name.

By and large, most big numerical codes are not written by numerical analysts, or even by computer scientists, but by engineers, physicists, and other large computer users. These people tend to underestimate the difficulty of producing reliable code, but in spite of this, or perhaps because of it, they have been responsible for most of the important methods that have been developed in the past (for example, most integration methods, the relaxation method, and the finite element method). When these people had a real problem to solve, they could not afford to be deterred by minor mathematical difficulties, so they invented new methods. There has been a tendency for numerical analysts and computer scientists to ignore or disparage the accomplishments of the writers of large problem-oriented packages--"just hack programming," although many major developments in our field have started with hack programming; the structured, polished programs and proofs have appeared

much later for distribution and publication. Many of the large codes utilize a fine blend of "engineering insight" and applicable theory to obtain results that could not be obtained by a numerical analyst or computer scientist. (In fact, one of the great difficulties facing us is how to codify such "insight" so that it can be applied to the development of general purpose methods and packages.) Part of the challenge of numerical software is to produce codes which can be embedded within large packages to handle standard operations such as the solution of differential equations. Although some modern numerical software is far more reliable than the corresponding sections of the large problem-oriented packages, these packages do not generally use library software because the latter is insufficiently flexible to be tailored to a particular class of applications and still retain efficiency. Unfortunately, many of us get too involved in our own theoretical interests to produce useful codes for the large body of users. We produce computer science trained programmers who are more interested in clever garbage collection than in avoiding generating functional garbage in the first place, or numerical analysts who hibernate in Hilbert space.

It is certainly not true that all numerical code written by the user has desirable properties. Vast numbers of small problems are "solved" everyday by ordinary users in ways that are not only painful to the theoretician, but which are wrong sufficiently often that we should be concerned. Regrettably, much of this code has found its way into computer libraries in the past, although not only is it not suitable for a public library, it is often too crude for most adult bookstores! The wrong answers arise because, as Shampine* has pointed out, crude numerical methods are often not adequate for solving crude numerical models. Whereas, when a user is faced with a very large job, time is invested to try out many methods and at least make an empirical choice, a user faced with a small job tends to choose the first simple method that appears to work--that is, which gives an answer close to one

* - many comments in this talk are taken from talks by others. In the case that they have not appeared in print, I will not give an actual reference, but will acknowledge the source of the wisdom.

expected. These results are incorrect for precisely those problems which are of more than normal interest to the user, namely those problems which have an unexpected behaviour not detected by a crude method. Thus the small user has a great need for reliable software which will solve large classes of problems. Efficiency is not as important for small problems because the human time presenting the problem to the computer is the more expensive resource. The desire for reliability is obvious; the need for broad applicability is less obvious but equally important, because, if a user has to select between a large set of codes on the basis of the characteristics of the methods and their application to the problem at hand, the choice made will most likely be wrong. It is unreasonable to expect a user to understand large bodies of knowledge in other areas, in this case in numerical analysis and computer science.

We see that some of the attributes needed in numerical software are reliability, efficiency, and broad applicability. Is there any reason why this type of code cannot be, or is not, written by numerical analysts? In a sense it is because the scientists/engineers/programmers are the people who could be said to be the true numerical analysts as they practice numerical analysis in the sense that they analyse problems numerically. However, the term "numerical analysis" has come to mean something different, the study of numerical methods themselves, and that is what today's numerical analyst does, mostly studying the methods originally developed by the engineer or scientist. Today's numerical analyst does not write code (if it can be avoided)--that is a job for a programmer; but not to write code is to ignore a very important step in the two stage activity we find in almost all intellectual endeavor; analysis and synthesis. In this case, we must analyse classes of problems and classes of methods, but before the result is of any utility, the result of the analysis must be synthesized into a computer code. A numerician is a person interested in both of these activities. The analysis is the science and the synthesis is the art, or alchemy. The result will be numerical software if properly done.

What I am really saying is that "Numerical software does it

better." Let me compare the situation with the the telephone system. Forty years ago the projection "everybody will be a telephone operator in thirty years" could be made on the basis of the growth of the system. In the last decade we could make the same projection about programming: "Everybody will be a programmer in twenty years." The fact of the matter is that everybody is a telephone operator today, at least in the western world. Almost everybody connects their own calls. Of course, they use a language that is high-level compared to the actual connections that have to be made. In the same way, everybody will be a programmer very shortly, but they will be using languages that are very high-level compared to what computer scientists think of as high level. These languages will allow access to the software tools that solve a range of problems automatically. The job of the numerician is to synthesize codes that can solve numerical programs automatically. (However, this is not to suggest that the solution to the sorts of problems I am concerned with is the design of yet another language. Far from it. Too many people are fond of designing new languages, languages which have wonderful structures for handling the " $n + 1/2$ loop" problem. What we have is not the " $n + 1/2$ loop" problem, but the " $n + 1/2$ language" problem. We already have n languages, and there is always another half-assed proposal being made.)

2. WHAT IS NUMERICAL SOFTWARE AND WHY IS IT DIFFICULT?

The previous remarks apply equally well to any form of software if we substitute "computer scientist" for "numerical analyst." What is special about numerical software other than that it deals with numbers? First and foremost, numerical software must tolerate errors. The word "error" is an unfortunate one because numerical errors are not errors in the usual sense of the word, but differences between approximations that can be computed in a finite length of time and the true solution. Wilkinson relates an incident in which he was visiting a univeristy to give a talk. At dinner the previous night he found himself sitting next to another guest of the university, a bishop. Opening conversation, the bishop enquired the subject of Wilkinson's talk. "Error," replied Wilkinson. "That's a coincidence," replied the bishop, "that's my topic

also, but I call it sin." Numerical error is unavoidable; it need not be sin.

Numerical software has two principle characteristics:

- (i) It deals with approximations to real numbers.
- (ii) It is usable on a range of computers which have different approximation capabilities.

The fact that it deals with approximations to the real numbers causes the dimension of all aspects of software production to increase. Non-numerical software deals with finite or countable sets, be they numbers or not. In it we are concerned with the design and analysis of an algorithm. Execution time studies are done on the algorithm. Memory space studies are done on the algorithm. Program proofs are prepared for the algorithm. The algorithm either "fits" into the computer, (that is, the range of integers and memory space is adequate), or it does not. If it fits, the analysis of the algorithm carries over to the behavior of the computer program. In numerical software, the characteristics of the algorithm are only one of the set of problems to be studied; the behavior of the actual implementation of the algorithm on a computer must also be analyzed.

The fact that computers differ in their treatment of floating-point numbers means that we must be concerned with classes of computers. As E. Battiste has pointed out, two decades ago we concerned ourselves mainly with the algorithm, a decade later we were also concerned about its embodiment on a particular computer, while today we are concerned about its embodiment on classes of computers in classes of languages. These classes of computers have different numeric ranges, different precisions, and different round-off properties. Numerical software is written to work in these varying environments. It must be sensitive to the precision of the computer so that it does not try to achieve more accuracy than is possible on a particular machine. It must be sensitive to the range of numbers so that it can avoid unnecessary overflows and underflows. It must be aware of the peculiarities of round-off. It must also work around the difficulties of many computer languages. A

good example of this is one given by J. Cody [2] in which he needed to compute $1.0 - X$ without any unnecessary rounding error. If X is in the range 0.5 to 1.0, this can be done without error on a computer that uses a guard digit during addition/subtraction. If not, it can be done by forming $(0.5 - X) + 0.5$. However, most optimizing compilers will "improve" this for the user--back to the original form. This can be circumvented in some systems by coding

```
T <- 0.5 - X
```

```
R <- T + 0.5
```

but a slightly better optimizing compiler will still oblige with the improvement. In many cases, the compiler can be outsmarted by adding a statement label to the second statement (I hope the pure computer scientists present will pardon such constructs). However, if the compiler does a flow analysis we are foiled again. Worse yet, we may get an error message "UNNECESSARY STATEMENT LABEL ON LINE nnn".

(Sometimes it seems that the system programmer cannot leave well enough alone. I am reminded of the story about the priest, lawyer, and system programmer waiting to go to the guillotine. The priest went first, was asked whether he wished to lie face up or face down, and chose face up to look at heaven. The blade fell and miraculously stopped a millimeter from his neck, so he went free. The lawyer went next, and, unwilling to break precedent, chose to lie the same way. Again the blade stopped a millimeter short. The system programmer went last, and chose face up because he was interested in examining the mechanism. "Ah," he said, "I see the problem. The rope isn't on the pulley correctly." The other viewpoint is seen in the probably true story of a programmer working for an aircraft company some twenty years ago. Observing that the square root routine was happily returning a value even when the argument was negative, he changed it so it would trap and warn the user. The inevitable result was that programs which had previously worked "perfectly" before now failed. Needless to say, the decision of the management was to restore the routine to its previous state in which it gave no unpleasant suggestion that all might not be well.)

Numerical software can be particularly difficult to design because,

even before precision, range, and round-off problems are considered, many numerical tasks are, in a sense, unsolvable. It is no use telling the user that the problem is theoretically impossible; it has to be "solved." A numerical program can come to one of three outcomes: answers correct to within the tolerance requested or expected by the user; a statement by the program that the task is impossible; or answers not within tolerance. The latter are more commonly called wrong answers. Ideally, we don't want wrong answers, and many users are prepared to ask that they not occur. However, the best we can usually ask is that we minimize the frequency of occurrence of wrong answers, even at the expense of telling the user that the job is impossible more frequently. It is not true that "some answer is better than none." In most cases, a wrong answer is much worse than no answer. Suppose the user thereby makes a wrong decision; it is clearly better not to build a plane than to build one that will not fly.

Why is it so difficult for numerical algorithms to distinguish between possible and impossible cases? In some cases it isn't; it depends on the type of problem. The types of numerical problems can be classified according to three criteria: data properties, algorithm properties, and round-off error properties. The first subdivision is on the basis of the initial data provided by the user to specify the problem. This data could take the form (in ascending order of difficulty for the computation):

- (1) A set of isolated values, such as the coefficients of a system of linear equations or the argument to a sine function.
- (2) Symbolic data, such as the specification of a differential equation.
- (3) A "black box" program which will compute any specific value of a function for specified values of its arguments.

The first two forms of data give a complete specification of the problem, although considerable (non-numeric) manipulation may be required if the data is in the second form. If the data is in the third form, we must accept that absolute reliability is impossible without

additional assumptions. For example, if we write a program to compute the value of the integral of a function $f(x)$ by forming a weighted sum of values of the function f for various values of its argument x , we will only sample f at a finite number of points, say x_1, x_2, \dots, x_n . We can substitute another function $g(x)$ which is identical to $f(x)$ at these points, and the integral of g will be "identical" (numerically) to that of f . For example, let

$$g(x) = f(x) + (x - x_1)^2(x - x_2)^2 \dots (x - x_n)^2$$

The second subdivision is on the basis of the relation between the algorithm and the problem, and is independent of round-off error problems. It leads to the breakdown into the four groups:

- (A) The problem can be solved by a finite sequence of calculations in real number arithmetic (that is, infinite precision arithmetic). Linear equation solution is an example of this.
- (B) The problem cannot be solved exactly in a finite sequence of arithmetic operations, but there exists one or more finite sets of steps for which everything needed to complete an error analysis and get error bounds is known. An example of this is a program for cosine.
- (C) Error bounds can be given in terms of unknown characteristics. These characteristics are values such as the bounds on derivatives which cannot be computed. Examples of this include the solution of the ordinary differential equation $y' = f(y)$ and the solution of the non-linear equation $f(x) = 0$, where f is a function given by another program.) Note that in these examples there is an assumption that certain derivatives exist and are bounded. These assumptions are usually correct but cannot be verified.
- (D) All known proofs of error bounds depend on assumptions that not only cannot be verified, but which are often not true. An example is a program for partial differential equations. Most theories rely on linearity or small deviations from linearity, but practical problems that do not satisfy these assumptions are often

solved.

Group (A) problems are trivial at the algorithm level, as there is no analytical problem to consider. Neither do group (B) problems cause difficulties at the algorithm level; a code can be designed using a number of steps determined by the accuracy needed. This is not to say that the implementation of group (A) and (B) problems on an actual computer is trivial; there are still the problems of precision, range, and round-off to consider. (These are discussed for a variety of functions in the previously cited paper of Cody.) However, the difficulties are soluble.

Group (C) problems are the first to exhibit serious difficulties. Much software allows the user to request answers within a given error tolerance. However, the error bounds that can be computed depend on unknown quantities. Although these quantities can be estimated, and that is what numerical software does, there can be no guarantee that these estimates are correct. Consequently, a wrong answer is always a possibility. Since a key objective of numerical software is reliability, the most the numerician can hope to do is to keep the probability of wrong answers low. If instead, we can "get hold" of the function by requiring the user to specify form (2) data, we might be able to move the problem into group (A) or (B). Alternatively, it is sometimes possible to convert a group (C) problem to one in group (B) by requiring the user to provide additional information. For example, in solving the equation $f(x)=0$ we could also ask the user to provide a subroutine which will give a bound on the derivative of $f(x)$ over a range of values of x . In that case, we can compute error bounds (if round-off error is ignored) and can write programs that make statements such as "there are no roots of $f(x)=0$ in the range specified."

It is only recently that numerical software has been attempted for problems in group (D), both because of the great difficulty in providing much reliability, and because it is still difficult to know how to handle many aspects of such problems. For example, many partial differential equations have to be solved in regions with very irregular boundaries. These give difficulties both in specification and in

numerical treatment. Consequently, we find a number of software packages on the market at the moment, each suited to a particular combination of equations types and boundary conditions. See, for example, the Ellpack project [10]. (Advances in mathematical understanding may cause group (D) problems to move to group (C).)

The third criterion by which we can categorize problems is by their dependence on round-off errors. This leads to a subdivision into the types:

- (a) Ones in which a priori bounds can be computed on the effects of round-off errors. This occurs, for example, if we wish to compute a cosine over a limited range of its argument.
- (b) Ones in which we can compute bounds on the effects of round-off errors once we know the data for a particular problem. Linear equations are a case of this if we want to determine the error in the answer.
- (c) Ones in which no bounds can be specified on the effects of round-off errors. This usually arises with problems in Groups (C) and (D) because the effect of the propagation of round-off error is dependent on unknown characteristics of the actual problem.

I have talked about the "effects" of round-off error without being very specific. By "effect," most users mean the change to the answer. Bounding the change to an answer is called forward error analysis because it computes the effect of the error as it propagates forward with the solution process. In this type of analysis, the error at the end of the calculation will depend on the way in which it is amplified or reduced by the problem and the solution process. If the problem is such that small errors are amplified greatly, the problem is called ill-conditioned, because small changes in the initial data, whether by error or user perturbation, cause large changes in the answers. An example of an ill-conditioned problem is the problem of computing the trajectory of a rocket with no guidance system fired from earth and aimed at Mars. A very small error will send it past Mars and probably into the sun, causing about a 100% error in the result! There is no way

of avoiding growth of errors if the problem is ill-conditioned. If the method is such that it causes small errors to be amplified even though the underlying problem does not, we say that the method is unstable. Unstable methods are to be avoided! The other type of error analysis that is very popular with numerical analysts is backward error analysis. This tries to determine the smallest change to the input data which could lead to the answer obtained. Expressed mathematically, we have a problem, say $P[d,x]$, where d is a set of input data and x is the unknown. Ideally, we would like to solve this, that is, to find a value y such that

$$P[d,y] = 0$$

Unfortunately, we compute a numerical answer z . In forward error analysis we try to determine the size of $z - y$, whereas in backward error analysis, we ask how big Δd has to be in order that

$$P[d + \Delta d, z] = 0$$

The advantage of backward error analysis is that it is often possible to get bounds on the backward error that are independent of the problem data, even if the problem is ill-conditioned. In the rocket example, it may happen that the rocket would actually hit Mars, but a numerical computation of the trajectory shows that the rocket will miss and crash into the sun. However, we can say that the computation produced the correct answer to a problem with very slightly different data. For a more down-to-earth example, suppose we want to compute the sine of a large number, say $\sin(10^6\pi)$. In a seven-digit precision computer, we will compute $\text{SIN}(3141592)$. Will this be zero? Of course not. Its correct value is about -0.608 , but with round-off error, we could get any answer between -1 and $+1$. Thus, a bound in forward error analysis will be of little value. However, a bound in backward error analysis will tell us that we have the sine of a number that is within one part in about 0.7×10^{-6} of the given argument because, even with the worst case error of 1.608 , we know that

$$\sin[(10^6 + 0.5)\pi] = 1.0$$

and

$$\frac{(10^6 + 0.5)\pi - 3141592}{3141592} = 0.708 \times 10^{-6}$$

Consequently, backward error analysis is a very appealing concept for the numerical analyst; it passes the buck back to the users who, after all, cannot expect good answers to bad problems. If we give them an answer to a very close problem, what more can they ask? Unfortunately, even in a simple example such as linear equations, the meaning of "very close" may depend heavily on the problem area. For example, a simple linear electrical network of resistors leads to a system of linear equations. It is possible to say for a reasonable code that the answer obtained is the answer to the system of equations changed by a small amount. However, to an electrical engineer, a "small change" means that the network differs from the original only by some small changes to resistor values. Unfortunately, the numerical analyst means that there may also be some additional small resistors, or that Kirchoff's laws are only satisfied approximately. The engineer may not agree that the new network is in any sense close to the original one!

It is important to realize that the classification of a problem is dependent on the demand the user places on the error. If a backward error bound is sufficient, the problem may be computationally much simpler. Thus, with a backward error bound, a linear equation problem is 1Aa (class 1, group A, type a), but with a forward error bound, it is 1Ab.

Earlier I said that a key attribute of software is reliability. How do we get this? Traditionally, testing has played a major role in checking for reliability; today, program proof techniques are taking a role in checking non-numerical software. Can they be applied to numerical software? Generally speaking, no, for a three reasons:

- (1) We can't prove theorems if we don't know what we want to prove.

- (ii) We can't prove theorems for algorithms if we can't even prove theorems for the underlying mathematical problem.
- (iii) Theorems have to be proved for computer implementations using inexact computer arithmetic.

The first statement is true for proofs of any subject matter! In terms of good programming practice, it is usually expressed in the statement "we should not write any line of code until we know exactly what we would like to prove about it." The difficulty with numerical problem solving is that we frequently don't know just what it is that can be proved. The user is inclined to say that the only reasonable result is a statement indicating the maximum error in the answers. This is, of course, just a forward error bound. In other cases, however, it is not a reasonable demand and a backward error bound is much preferable. More serious is the fact that in many real problems, we have no idea what is meant by error. The following example was given by A. Erisman. An engineer was examining some results provided by the friendly local computer center staff, and comparing them with the true solution for a case in which the latter was known. Figure 1 below shows the two "solutions", the numerical one is dashed. The computer center staff were somewhat discouraged by the apparent difference between the two solutions. Fortunately, the engineer was quite happy with the results. "It has about the same number of oscillations, they damp out in about the same way, and they reach the same asymptotic value," was his response. Mathematicians do not know how to measure error so that the two solutions shown are close, and until we know that, we can not hope to prove theorems about such programs.

We can't prove theorems for general classes of problems about methods that do not work for some undetermined subclass of those problems. This means that we can forget about proving results for problems in group (D), and the most we can hope to do for group (C) is to prove results in the presence of additional assumptions which can only be tested rather than verified. Note that this means that even when we have accepted the idea that the program may give "no answer"--that is, it may return with the message "I can't solve this

problem," we still can't expect to prove that the program never lies if the problem is in group (C) or (D). Thus we are left with the possibility of proving results for the first two groups. Examination of codes for problems in these groups reveals that they are relatively simple logically. The complexity arises in the related mathematics. (For example, the fact that a code to solve linear equations will work may depend on the positive definiteness of a matrix, a fact which will imply some algebraic identities which prevent overflow or divide by zero.) Because group (A) and (B) algorithms are not complex, the utility of proof techniques for codes in the first two groups is low; about the best they can hope to do is to provide a mechanical verification of the steps in a mathematical proof and to check that the code matches those steps.

If we want to prove results for actual codes, we must be able to make precise statements about the properties of floating-point arithmetic when such properties are used in a program. Such statements are beyond the capability of most computer manufacturers--take a look at their machine description manuals; the only way to conclude that IBM manuals are well written is to read a CDC manual. Either the manufacturers are hopelessly incompetent when it comes to description or it is very difficult to describe the design. If it is the latter case, as seems likely, there is little hope of achieving a specification useful for an automatic theorem prover such as we might use to verify programs.

In many cases, human analysis of the program is done using less stringent approximations to the rounding errors than actually occur. Then, it is possible to prove results about codes for problems in groups (A) and (B). An example of this is given by Dekker [4]. If the floating-point unit can be assured to meet reasonable specifications, some proofs are possible. Dekker examines the problem of finding two values y and z such that do not differ by more than a given tolerance and for which $f(y)$ and $f(z)$ have opposite signs. If $f(x)$ is continuous, this gives a zero of f within the specified tolerance. (Initial values for which $f(x)$ has opposite signs are given, so this is a group (A) or

group (B) problem, depending on your point of view.) Brown [1] has constructed a model of floating-point computation. The parameters of the model are chosen so that anything that can be done by the model can be done by the machine being modelled. Furthermore, the model is "clean" so it can be characterized by very few parameters. However, this approach leaves open the question of proving that the approximations made to the rounding errors are met by the computer!

Additional complications arise when underflow and overflow are considered. Non-numerical problems do not encounter underflow, and they can handle overflow by going to increased precision using a multiple precision package. Numerical software should operate whenever possible, that is, it should operate if the input and answers are in range. For example, a piece of mathematical software cannot in general include an expression such as $\text{SQRT}(X^{**2} + Y^{**2})$. If all exponents are equally likely, the answer is in range almost always, but intermediate results overflow about 44% of the time (if exponent of either X or Y exceeds half the maximum), and underflow about 6% of the time (if both exponents are less than half the minimum). See Figure 2. (It is true that these are Madison Avenue statistics; exponents are not equally distributed, fortunately, but that does make the problem go away.) This means that even simple calculations can become quite contorted if they are to be generally applicable.

Thus, we see that a large part of the difficulty of numerical software is due to the lack of a scientific basis. There is no simple criterion which must be optimized, but a set of ill-defined goals such as reliability, generality, and utility which we have so far failed to quantify. Consequently, a large part of the work of a numerician is art rather than science.

3. THE SCIENCE

When software is built, various techniques are used to analyze the problem and design the code. The numerician is not only concerned with the usual analysis for speed and space, but with:

- (i) Stability analysis
- (ii) Asymptotic analysis
- (iii) Round-off error analysis

These use standard techniques from numerical analysis. First I want to take a brief look at the ideas in such analyses. Then, in the next section, we will see that they are based on assumptions that are not true in many cases. This is when the art of the numerician is needed.

Suppose we have a problem to solve. Let us ignore input parameters and write it as

$$P[y] = 0$$

that is, the answer is y , and it satisfies some relationship P . Unless the problem is in group (A), the algorithm to compute an approximation z to y takes some other form, say

$$A[z]=0$$

because P involves operations such as differentiation that do not exist in the computer. This "implicit form" may seem like a strange way to express an algorithm which is, by definition, an explicit formulation of a way to compute the result. However, for many problems, it is relatively easy to devise an expression of the form $A[z] = 0$ that is approximately satisfied by the true solution and which can be solved explicitly. For example, if we are given the differential equation

$$\frac{dy}{dx} - f(x,y) = 0$$

we can replace the derivative by an difference approximation to get

$$\frac{z(x+h) - z(x)}{h} - f(x,z) = 0$$

which can be solved explicitly for a sequence of values $z(x_0 + nh)$ given $z(x_0)$. Naturally, we expect z to be close to y . We hope to achieve this by making A "like" P . There are several ways of asking whether A

is like P . The two principal ones are

residuals

and truncation error

In the residual approach, we ask whether the numerical solution z comes close to "satisfying" the problem P , that is, whether $P[z]$ is small. In the truncation error approach we ask if the true solution y comes close to satisfying the algorithm or code, that is, whether $A[y]$ is small.

The definitions of residual and truncation error are

$$\text{residual } r = P[z]$$

$$\text{truncation error } t = A[y]$$

By subtracting $P[y]$ and $A[z]$ (both of which are zero) from these, we get

$$r = P[z] - P[y] = \frac{\partial P}{\partial x}(z - y)$$

$$t = A[y] - A[z] = \frac{\partial A}{\partial x}(y - z)$$

If we can invert the partial derivatives (which are matrices), we get

$$z - y = \left| \frac{\partial P}{\partial x} \right|^{-1} r$$

and

$$z - y = \left| \frac{\partial A}{\partial x} \right|^{-1} t$$

The first says that the error in the answer ($z - y$) is small if the residual is small and the quantity $\left| \frac{\partial P}{\partial x} \right|^{-1}$ is small. The latter quantity depends on the problem, and we say that the problem is well-conditioned if it is small. There are some problems, usually in groups (A) and (B), for which we can calculate a residual (for example, in linear equation solution). In those cases, we can compute an error bound on the result if we have a problem whose condition we know. For example, in the linear equation $Py = b$, we can, in principle, compute the residual of the solution z by forming $r = Pz - b$. If we do this in extended precision, we can ignore round-off errors in the residual computation.

Combining this with $P y - b = 0$ we get $z - y = P^{-1}r$. Thus, in this case we can compute both the residual and the condition of the problem. In other problems, particularly in groups (C) and (D), we cannot calculate a residual because P involves operations such as differentiation which cannot be done exactly when the values of functions are available only on discrete points. In that case, we usually use the truncation error approach, and say that the solution error is small if the truncation error is small and the quantity $[\frac{\partial A}{\partial x}]^{-1}$ is small. If the latter quantity is small, we say that the method is stable. This is the type of analysis applied to differential equations. In the example given above, which happens to be Euler's method, the simplest method for solving differential equations, we can substitute the true solution y into the method and find that the truncation error is $h d^2 y / dx^2 / 2$ where the second derivative is evaluated at some unknown point on the solution. In this case, we can show that the method is stable if the original problem is well-conditioned, so that the method is a good one.

If something is known about the condition of the problem, it is possible to say something about the accuracy of the answer when it is possible to compute the residual. However, it may not be easy to see how to create an algorithm with small residuals. On the other hand, it is frequently easy to see how to create an algorithm with small truncation error; then it is much more difficult to make the algorithm stable.

Truncation errors are often analyzed using the techniques of asymptotic analysis. Essentially, these are applications of Taylor's series to appropriate expressions. When it is impossible to do an exact computation for P , even in the absence of round-off error, we usually have an algorithm which depends on a parameter. Suppose this parameter is a small number, say h , and the algorithm is

$$A[h, z] = 0$$

For many problems, particularly those arising in differential equations, the algorithm run time depends on h , and becomes infinite as h approaches zero. On the other hand, the algorithm becomes more accurate

as h reduces because it has been chosen so that it is exact when h is zero, that is, so that $A[0,y] = 0$. Then we can compute $A[h,y]$ by Taylor's series to get

$$A[h,y] = A[0,y] + h \frac{\partial A}{\partial h} + \frac{h^2}{2} \frac{\partial^2 A}{\partial h^2} + \dots$$

We already have arranged to make the first term zero. The remaining terms are the truncation error. Usually we design an algorithm to make a number of the additional terms zero. Suppose all of the terms up to h^{p-1} are zero. Then, for small enough h the error is close to the first non-zero term which is proportional to h^p . Numerical software frequently relies on estimating this term to measure the error and to control h . It ignores higher-order terms.

Traditional floating-point error analysis allows the error introduced in each step in the computation to be related to the size of the numbers taking part in that computation. Thus, the initial errors are proportional to the initial data, and subsequent errors are related to the intermediate data values. For this reason, it is often straightforward to relate the effect of round-off errors to the effect of changes in the initial data, since these tend to propagate proportionally through each intermediate result. That is why backward error analysis is such a handy tool. For example, in a simple calculation such as $B*(C + D*E)$, the value of the roundoff error in the result is a complex expression, whereas it is easy to say that the error in the result is no worse than that caused by a small change in the input values.

4. THE ALCHEMY

The creation of numerical software is based on the type of analysis discussed in the previous section, but none of this analysis is strictly valid in the real world of computers.

- (1) Stability analysis may be invalid because it depends on the differentiation of codes, not algorithms.

- (ii) Asymptotic analysis may break down because often we cannot be certain we are working with small values of the parameter--in fact, sometimes we know we want to use large values.
- (iii) Round-off error analysis breaks down because, in the presence of underflow, it is more complex than the usual analysis indicates.

Most of the time the analysis is not badly in error, but since the goal of a numerical software project is reliability, the success is related to the probability of avoiding wrong answers. An occasional breakdown of the analysis may cause a large code to give misleading answers frequently.

The differentiation of programs obviously fails in the presence of significant round-off errors, but fortunately round-off errors are frequently small compared to truncation errors. However, differentiation can also fail because many programs are adaptive; that is, they try to adjust some parameters to achieve close to optimal behavior. If these parameters behave in an erratic way or are confined to discrete values as would happen if they represented switching between various methods, differentiation appears to have no meaning, even in an approximate sense. This means that a totally different approach is needed for the analysis of stability. There has been little progress in this area yet.

Asymptotic analysis also fails when differentiation is not meaningful, but the major practical difficulties with asymptotic analysis seem to arise from the "small h " assumption. When h is not small, error estimates are unreliable if not totally wrong. Because of this, some of the toughest problems are those in which we want very little accuracy. As contradictory as it may seem, we know how to solve many problems very accurately, but we don't know how to solve them inaccurately and cheaply. In some cases this is simply the statement that the first digit is the most expensive to obtain, and others become progressively less expensive, but in other cases, we just cannot find ways to compute low accuracy answers reliably. This is particularly the case in differential equations where the only form of error estimates

known to us are based on asymptotic error estimates that break down at low accuracy. Consequently we have no idea of the size of the error until we have computed a very accurate answer. An alternate theory to asymptotic analysis is needed. Approximation theory provides a basis for some cases, but it has not been applied successfully to many problems which currently use asymptotic analysis.

Asymptotic analysis has another peculiarity in that if we use it to estimate an error we finish up with no error estimate! The reason for this is that if we estimate the error in a numerical result, we naturally subtract that error from the result to get a "more accurate" answer. However, we now have no error estimate for the more accurate answer. Kahan argues that we should solve this dilemma by computing an "uncertainty" rather than an error estimate. The uncertainty would represent the possible change to the solution due to our lack of more precise knowledge of the input or unwillingness to compute more precise information about the solution.

The breakdown of simple round-off error analysis can be seen in the example used earlier, $B*(C + D*E)$. Suppose that this is computed in floating point using a finite exponent range. Overflow is not a serious problem in that we are told when overflow occurs, and it warns us of trouble. We could take the same attitude to underflow, but most users ignore underflows entirely, accepting a zero result. Most of the time this is reasonable. However, suppose that C is the smallest number that can be represented in the machine, and B is its inverse, assumed representable. Let D and E be small enough that $D*E$ is just less than C , so is underflowed. Then, the computed answer for the above expression is 1.0, whereas the correct answer is almost 2.0. Mention should be made of an effort underway by an IEEE subcommittee on microprocessor floating-point standards. In addition to standardizing formats, they are considering two proposals which would help with this underflow problem. One would virtually eliminate underflows and overflows by using system traps to extend the range of the exponent when either occur. A heap would be maintained to keep this additional information, and bit patterns in the data would be used to indicate

extended exponent range. Arithmetic is likely to be slow for extended range numbers in this scheme, but some of the problems facing the numerician disappear. The other proposal introduces the idea of "gradual underflow." This allows numbers with the minimum exponent to be unnormalized. The effect of this is that if expressions such as the one above have a non-zero value, they will not be in error by more than one expects in a conventional error analysis [3]. In the past a number of schemes have been tried to minimize the impact of round-off errors. Some of these have been designed into computers but have not been effective. For example, significant-digit arithmetic [8] was tried about 20 years ago. The idea was to keep as many digits in the floating-point mantissa as were known to be correct. While the scheme did not give answers that were wrong, it did give answers with no precision left because most schemes for bounding the effects of round-off errors lead to answers that are far too pessimistic. A more recent scheme that has had some success is interval arithmetic [9] in which an interval known to contain the result is computed. Schemes of this form are, unfortunately, limited to problem that are not in class (3) or in groups (C) or (D), so round-off errors will remain a problem for the numerician for a long time.

5. CONCLUSION

What is the future direction of numerical software? Today we see a large amount of activity in areas that will formalize the measurement of methods, the selection of algorithms, and and the techniques of testing. For example, Jackson et al [6] have used a model problem and selected methods that are optimal over a class of model problems. Rice [11] has examined the algorithm selection problem as an approximation problem. A recent IFIP working conference [5] was devoted to the question of evaluation of the performance of numerical software. We might ask whether numerical software is a part of computer science and should be studied in computer science departments. I believe that computer science is first and foremost concerned with the analysis and synthesis of the design and use of computers. From this point of view, I place program design, programming language design, and architecture at the

center of computer science. Abstract theory, while vitally important and interesting in its own right, is only computer science when it is concerned with real computer problems. From this point of view, numerical software is very much a part of computer science. It is concerned with automatic problem solving, that is, with analysing methods and synthesizing techniques. It has some solid scientific foundations, but still involves a lot of judgement because the idealized goals are not achievable. Thus, the numerician is partially reduced to alchemy; brewing new concoctions and testing the results. Sometimes heuristics are used (although they are usually called adaptive methods and given something of a scientific footing). Many people have commented that the goals of automatic problem solving and the use of heuristics suggests a potential affinity with the AI community. Perhaps that will be true because our AI colleagues are the outstanding alchemists of computer science. However, the numerician is usually more concerned with writing a package that works reasonably well over a broad spectrum of problems than with a package that does remarkably well for a smaller set of problems. Perhaps that will cause you to accuse us of a search for the mediocre. If, by that you mean we are trying to build the Model A for everybody rather than the custom design for the few--then yes, that is what we are trying to do. We want a model that runs reliably, smoothly, and can be maintained easily.

ACKNOWLEDGEMENT

The thoughts presented here are the result of listening to numerous of my colleagues. Unfortunately, I cannot always remember who was responsible for which idea, so I have not always acknowledged them, but I would like to specifically thank Rob Skeel of the University of Illinois for his many comments and suggestions.

Bibliography

- [1] Brown, W. S., A realistic model of floating-point computation, in Mathematical Software III, pp 343-360. ed. J. R. Rice, Academic Press, New York, 1977.
- [2] Cody, W. J. "Software for Elementary Functions," in Mathematical Software, ed J. R. Rice, Academic Press, New York, 1971. pp 171-186.

- [3] Coonen, J. T., "Specifications for a Proposed Standard for Floating-point Arithmetic," Revised memorandum # UCB/ERL M78/72, Dec 6, 1978, Univ. of California at Berkeley, Department of Mathematics.
- [4] Dekker, T. J. "Correctness Proof and Machine Arithmetic," to appear in Proceedings of IFIP Working Conference on "Performance Evaluation of Mathematical Software," North Holland Press.
- [5] Fosdick, L., ed. Proceedings of IFIP Working Conference on the Performance Evaluation of Numerical Software, December, 1978, Baden, Austria. North Holland Press, Amsterdam, to appear.
- [6] Jackson, K. R., Enright, W. H., and Hull, T. F., "A theoretical Criterion for Comparing Runge Kutta Methods," SIAM Journ. Numerical Analysis, 15, #3, June 1978, pp 618-641.
- [7] Jacobs, D., ed., Numerical Software: Needs and Availability. Academic Press, N. Y. 1978.
- [8] Metropolis, N., Ashenurst, R. L., "Significant Digit Computer Arithmetic," IRE Trans on Electronic Computers, vol EC-7, 1958, pp265-267.
- [9] Moore, R., Interval Arithmetic, Prentice-Hall, New Jersey, 1966.
- [10] Rice, J. R., "ELLPACK: A Research Tool for Elliptic Partial Differential Equations Software," in Mathematical Software III, ed. J. R. Rice, Academic Press, New York, 1977.
- [11] Rice, J. R., "The Algorithm Selection Problem," in Advances in Computers, 15, ed. Rubicoff and Yovits, Academic Press, N.Y. 1976.

Figure 1. Computed solution versus actual solution

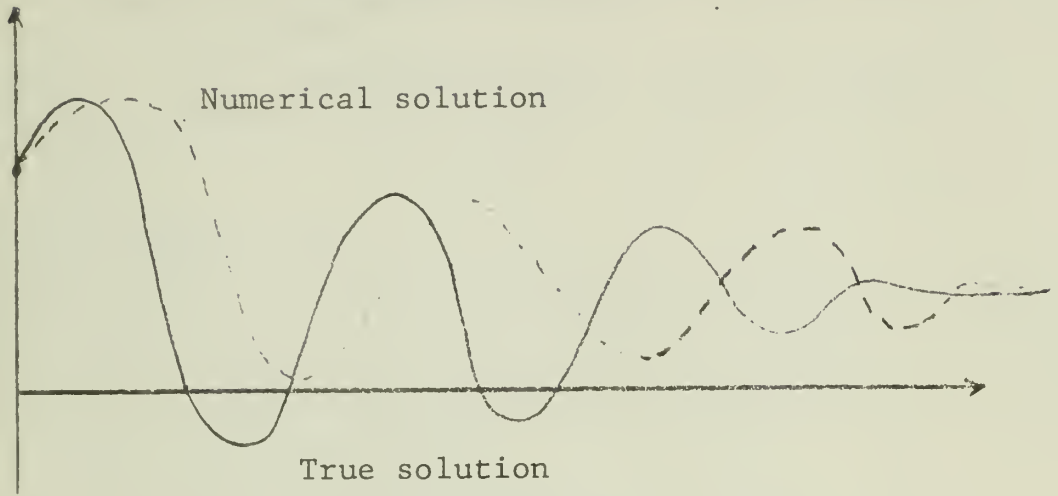
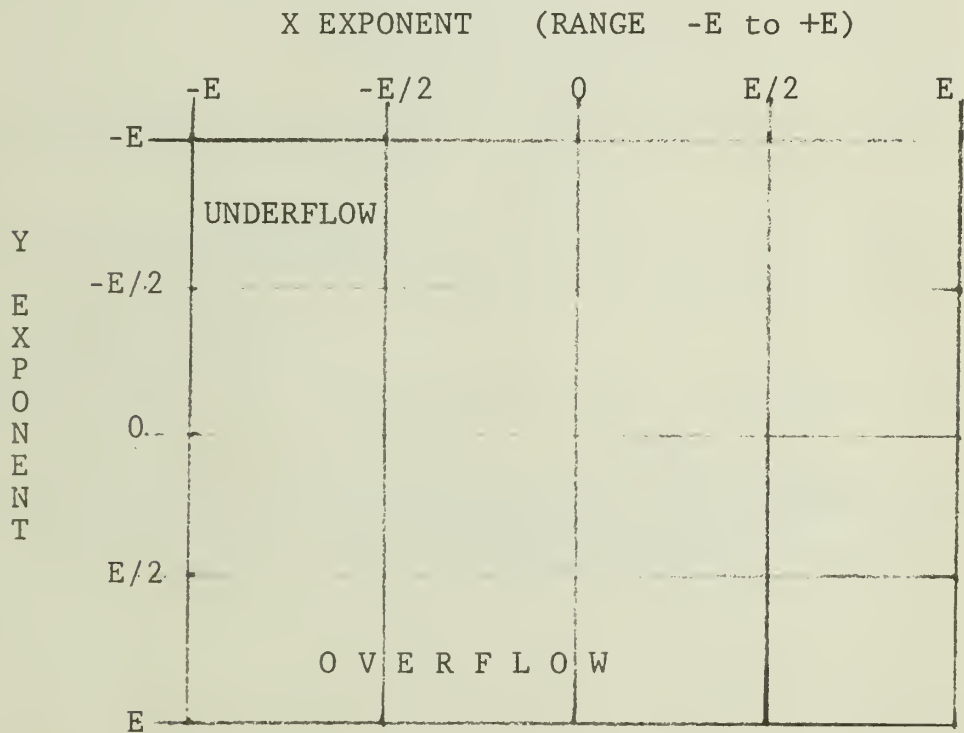


Figure 2. Overflow/underflow failures in $\text{SQRT}(X^{**2} + Y^{**2})$



U. S. ATOMIC ENERGY COMMISSION
UNIVERSITY-TYPE CONTRACTOR'S RECOMMENDATION FOR
DISPOSITION OF SCIENTIFIC AND TECHNICAL DOCUMENT

(See Instructions on Reverse Side)

1. AEC REPORT NO.

COO-2383-0059

2. TITLE

Numerical Software: Science or Alchemy?

3. TYPE OF DOCUMENT (Check one):

- ☒ a. Scientific and technical report
☐ b. Conference paper not to be published in a journal:
 Title of conference _____
 Date of conference _____
 Exact location of conference _____
 Sponsoring organization _____
☐ c. Other (Specify) _____

4. RECOMMENDED ANNOUNCEMENT AND DISTRIBUTION (Check one):

- ☒ a. AEC's normal announcement and distribution procedures may be followed.
☐ b. Make available only within AEC and to AEC contractors and other U.S. Government agencies and their contractors.
☐ c. Make no announcement or distribution.

5. REASON FOR RECOMMENDED RESTRICTIONS:

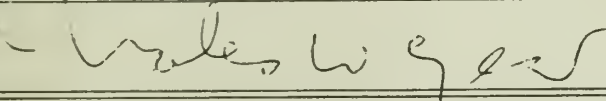
6. SUBMITTED BY: NAME AND POSITION (Please print or type)

C. W. Gear
Professor and Principal Investigator

Organization

Department of Computer Science
University of Illinois U-C
Urbana, Illinois 61801

Signature



Date

June 1979

FOR AEC USE ONLY

7. AEC CONTRACT ADMINISTRATOR'S COMMENTS, IF ANY, ON ABOVE ANNOUNCEMENT AND DISTRIBUTION RECOMMENDATION:

8. PATENT CLEARANCE:

- ☐ a. AEC patent clearance has been granted by responsible AEC patent group.
☐ b. Report has been sent to responsible AEC patent group for clearance.
☐ c. Patent clearance not required.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-79-969	2.	3. Recipient's Accession No.
	4. Title and Subtitle Numerical Software: Science or Alchemy?		5. Report Date June 1979
7. Author(s) C. W. Gear		8. Performing Organization Rept. No. UIUCDCS-R-79-969	
9. Performing Organization Name and Address Department of Computer Science University of Illinois U-C Urbana, Illinois 61801		10. Project/Task/Work Unit No.	
		11. Contract/Grant No. ENERGY/EY-76-S-02-2383	
12. Sponsoring Organization Name and Address Department of Energy Washington, DC		13. Type of Report & Period Covered Forsythe Lecture	
		14.	
15. Supplementary Notes			
16. Abstracts This is a summary of the Forsythe lecture presented at the Computer Science Conference, Dayton, Ohio, in February 1979. It examines the activity called "Numerical Software," first to see what distinguishes numerical software from any other form of software and why numerical software is so much more difficult. Then it examines the scientific basis of such software and discusses what is lacking in that basis.			
17. Key Words and Document Analysis. 17a. Descriptors numerical software library software			
17b. Identifiers Open-Ended Terms			
17c. COSATI Field/Group			
18. Availability Statement unlimited		19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 25
		20. Security Class (This Page) UNCLASSIFIED	22. Price

AUG 18 1980



UNIVERSITY OF ILLINOIS-URBANA



3 0112 047404956